Big Fish games

mohan.g.srinivasan@gmail.com

Wh@13M0h445!

restful-booker.herokuapp.com

Exploring Service APIs Through Test Automation.

**In this course discussing the importance of service API testing.**

What exactly is a service API,

Why is it so important to make sure that they have their own thorough test cycle?

Tools for API discovery and exploration,

see exactly which APIs your products are using.

Deeper dive into the Postman tool to dig into those APIs.

P.O.I.S.E.D. testing heuristic,

**P**reparing the test environment

**O**utput

**I**nteroperability

**S**ecurity

**E**xceptions

**D**ata

so you can improve and expand your test strategy.

How you can use Postman

to run automated tests,

do data-driven testing

and include your tests in a continuous integration (CI) environment.

**Lesson 1: Intro to API Testing**

* What is an API
* Who is the end user of an API?
* Why is it important to test the API?
* Tools for viewing APIs— Web Dev Tools (Chrome and Firefox) and Proxies

**Lesson 2: Postman Tool**

* Postman Installation
* Basic Requests
* Importing Requests
* Creating Collections

**Lesson 3: API Test Strategy (POISED)**

* 3.0 Preparing the test environment
* 3.1 Parameters
* 3.2 Output
* 3.3 Interoperability
* 3.4 Security
* 3.5 Exceptions
* 3.6 Data

**Lesson 4: API Automation in Postman**

* 4.1 Basic tests
* 4.2 Data-driven tests
* 4.3 Advanced Tests
* 4.4 Continuous Integration with Newman

**1 Intro to API Testing**

**A.P.I.** - **A**pplication **P**rogramming **I**nterface,

**one piece of code talking to another piece of code.**

website talking to a database back end,

a mobile application talking to an authentication service.

all sorts of configurations that it could be

**The way that an API works is through a contract.**

**A Request,**

Endpoint

Headers

Body

When the service receives this request,

**Response** (own set)

Status code

Headers

Body

**Let's look at an example of how this works.**

Here we are on Amazon's' website, and let's say we're searching for something, it could be anything. Let's take a look at what's happening in the world of API's.

So, I've narrowed it down to a single API, the “Completion” which is actually called every time you type a letter into the search box. This has all the elements we were talking about.

* We have the endpoint which is “suggestions.”
* We have various parameters that are being sent along. This is the interesting bit here, prefix B.
* And then we have a response. The response being its various suggestions for when I typed in B. Bluetooth headphones, et cetera.
* We also have the headers, which are pretty basic, and the status code which was OK that means that the request was successful.

And if we go to the next instance of this API call, it's almost the same, but now in the parameters we have “BA.” And the response is BA things, bath bombs, baby wipes, etc.

And it keeps going with our parameter changing each time, which for our final search term it was “bag,” so now it's doing bags.

This is just a single example of an API in the wild, on Amazon, that's performing a service for Amazon customers to help them find what they want.

**Even though the customer is the person who ultimately benefits from the API, in that case, it's important to remember that**

**the end user for an API is actually the developer of an application.**

The developers are the ones who are using the APIs to add functionality to their web pages, or their mobile applications, or whatever.

**Things that are important about APIs for developers are:**

**the APIs are**

* + **correct**
    - * + the contract is honored in the way that it's supposed to be.
  + **provides value** - doing something that the application needs;
    - * to show information to the customer
      * process data
* **intuitive**
  + - * the API should be easy for a developer to work with.   
        In the same way that a good UI will be easy for a customer to navigate;
      * a good API should also be intuitive for developers to program with.

**APIs are essentially public information.**

Even though the Completion API that we saw on the Amazon site was meant for the developers to use, we, the public, were able to go and look at it. We were able to see

the URL,

and the parameters that were used,

and we could **potentially call that API outside** of the Amazon website.

This is **true for any API that's exposed** through

a web application,

or a mobile application.

So, when you're testing your APIs, you can Not assume that they'll always be accessed through the expected user interface.

They could potentially be accessed from anywhere.

**Many companies**

**i.e. Twitter, Facebook and Spotify,**

**have publicly documented APIs**

**to encourage developers to create applications**

**that use their products.**

i.e. the Spotify API is very well documented.

*https://developer.spotify.com/documentation/web-api/reference/search/search*

It shows the endpoints, search, and the request parameters. There are also instructions on how to use them. This is a really good place to look at how an API could work, and all the different parts. And they actually have a way that you can even try them.

I would definitely recommend looking at the Spotify documentation as an example of a real API and how all the different parts come together.

**tools that will help you view APIs in various contexts.**

1 view APIs in your browser,

in Chrome, “View|Developer|Developer Tools.”

in Firefox, “Tools|Web Developer|Toggle Tools.”

These tool sets are both very similar

**Viewing API in Chrome Browser**

So here we are in Chrome. We just go to View|Developer|Developer Tools to bring the console up. Be sure to select the Network tab, because APIs are communicating over the network, then select XHR, which is for http requests. That's what our API requests are, http. And now we can see that when we type in the search box, our suggestions from Amazon come through.

We have headers, you can preview the request parameters, and you can see the response—all the stuff we were looking at before. You can also change where it is, or you can maybe get into a separate window. There’s timings, and all sorts of other tools within the Chrome developer tools.

#

**Viewing API in Firefox Browser**

Firefox has very similar tool set, which we saw before: Tools|Web Developer|Toggle Tools. We see the same options again: Network and XHR. As you type things in, they appear. And you can see pretty much the same information.

I find Firefox to be easier to read, it's a little easier on the eyes, and seems to have better response formatting, but both sets of tools are essentially the same functionality, in both Chrome and Firefox.

**Viewing API in Mobile Applications**

If you'd like to view APIs on your mobile applications, it's a little more complicated. You need to use a proxy tool, which is essentially something that will stand between your device and the rest of the internet and collect the traffic, and then display it to you.

two main proxy tools

Fiddler https://www.telerik.com/fiddler

owned by Telerik

optimized for windows.

free

Charles Proxy. https://www.charlesproxy.com/download/

works on Windows or Mac,

a license fee of about $50.

I use a Mac, so I prefer Charles Proxy, and if you can afford the license fee, or if your company will pay for it, I would definitely recommend Charles Proxy, as a proxy tool.

Just to show the kind of information you can get, out of out a tool such as Charles, this is something I captured from a game I was testing.

It shows all the endpoints.

We can also access scores and the levels,

and it shows the requests, and the response, and all the headers as well.

We can go to the sequence,

and see what order they were given in.

So, this tool is really, really great for discovering about your mobile APIs.

APIs are contracts that enable communication between different programs, so, the end result of an API is that

communication is happening

between two different services,

between two different applications,

between an application and a service.

**The goal is to have that communication happen,**

**and**

**the goal when you're testing - make sure that the communication is happening properly**

**2 Exploring APIs with Postman**

**Overview: In this chapter we will go over how to install Postman, how you can create requests or import them from other sources, and how to organize your requests in collections.**

**Note:**

For maximum learning experience, please view the course video for dynamic demonstrations.

So now that we've talked about what APIs are and we've looked at how we can explore them in websites and mobile devices, let's look at a tool that will help you really play with APIs and explore what they can do and what they can't do.

Postman tool, the best REST client on the market today.

To install it go to

https://www.getpostman.com/apps.

Once there, choose the correct executable for your platform, download it and it will install very easily.

**Once you open Postman, you'll see a form for entering URL and parameter information.** For example, if I type in this URL

***http://worldtimeapi.org/api/timezone***

for getting time zone information

I press [Send]

I get this list of time zones

Then if I take my own time zone,

***http://worldtimeapi.org/api/timezoneAmerica/Los\_Angeles***

put it in and now I get the time information. So this is an API that's available for getting the time and we can access it through Postman.

So once I do that, I can go to history. I can open up previous requests and send them again.

I can see what happens if I do something a little different,

***http://worldtimeapi.org/api/timezoneAmerica/Los\_Angele***

see what the errors are. I have the status information, the size of the download, the amount of time it took, all the information I need to better understand my API.

So that's the most basic functionality of Postman.

**Now that we've done some basic data entry, let's combine Postman with the web developer tools we were looking at before.** Here we are on the Amazon site. We have our completion requests and so we just have to select one, right click to bring up the context menu and do “Copy as cURL” (or on Windows,

“Copy as cURL (bash)”

and this works the same in both Firefox and Chrome.

copied it as cURL

go back to Postman

click Import (looks like the button's disabled, but it's not)

click raw text tab

paste the cURL

in Postman, the very same request.

If we send it we can see that it's working like the one on the website.

I have all these suggestions for things that start with, in this case “C”, because that's our prefix

can unclick query parameters;

and see if it still works.

they're tagged on at the end of the URL.

**using Postman with a GET API - just getting information**

“GET” - getting information through an API,

the parameters are often passed as part of the URL as query parameters.

So we can remove them here, and you can see that they are disappearing as I unclick and then we can send again to see if it works, and it's still working.

So after some experimentation I can see that only three of these parameters are actually necessary to make the request work the way we want it where it's returning suggestions. And although it only has these three in the URL, we still have a little bit of clutter here with our things that we don't want. So we can bulk edit. You can see here they are here, and we can just get rid of these. Save again and now we have a nice clean URL and parameters that we can experiment with more completely, and it's still working.

**Through a POST request - to actually do something with our API that's changing things on the server**

and let's look at an example of getting that from the browser. So we come back here to Firefox and now I'm at the [Big Fish Games website] (https://www.bigfishgames.com/) and I'm going to sign in. So now I'm going to sign in and when I press sign in, oh I have to clear my filter and here it is. It's a POST request this time because we're actually doing something not just getting information. So let's go ahead and copy as cURL and into Postman.

Here we have our API service living here at

***https://susi.bigfishgames.com/sallig/signin***

the endpoint is ***signin***

because we're signing in with an existing account. Here's the body of the request. This time it's not in the URL, it's part of a form.

So it's saved here as

x-www-form-urlencoded

and when we send this ...

Oops, we have a little issue where we have double encoding: blurb%40blop.com.

So the encoding is because certain characters can't be sent through the browser, they have to be changed into a special encoding. So the at sign (@) is one of those characters, and it's code is %40,

but we're going to change it back to an at sign

blurb@blop.com

and then we'll try this again.

All right, so now it worked and we can see the response. We have our user email, and the use her name and this is the token that we got back. So this is how we can do a post request with Postman.

**the final resource: the Restful Booker API Playground**

created by Mark Wintringham,

https://restful-booker.herokuapp.com/

Restful-Booker is a resource, it's a very simple API, it has some bugs already built into it and it's really great for practicing your API testing techniques.

If we come here to the API docs, we can see that it's very well documented, and again we have our little cURL request. So this is going to make it quite easy for us to create a collection of Restful Booker API calls that we can then in future lessons do our test passes on.

Once again we're going to Copy as cURL and put it into our Postman. Import here, and now we have our authorization request with our username and password. We're going to save this to a new collection. Let's just create a new collection here. We'll call it “Restful Booker” and now we can save this as “Auth” in Restful-Booker, and here we go. We're starting to build our broad collection.

So I've gathered all the requests from Restful Booker into our Postman Restful Booker collection along with the auth requests. We have several varieties of GET requests, which are getting by the name or the date or the ID. We have a POST request to create a booking. We can update a booking, either with the full booking information or with partial booking information, and we can delete.

You'll see that we have headers as well. If you recall, we have this Auth, this very secure Auth which will give us this token in response. Tis token is actually what is going into the header information. We haven't done much with headers, but here we can see that there's this cookie header that wants this token. So if we just try to do this with the default (token = abc123) it'll say it's forbidden, but if we put in the token for our Auth, now it works.

The POST request is creating a new thing, and the response is going to be different every time because it's returning a different ID. Let's look at how that's gonna work.

We'll change the name a little bit. So here we have this one coming back as 13, 14, 15. So we're actually creating resources, whereas with updating we can do this again and again and the response is always going to be the same because we're updating the same resource. So every time you send this response you can send it multiple times and always get the same response back, whereas with a POST you send it multiple times, you could get a different response back.

Iif you'll notice for a PUT, we need to have the whole object passed in just to change the name, but for a PATCH we can just input the things that we want to change. So here again, and it returns the entire object this time with just what we wanted changed and then DEL is going to delete the object altogether.

The great thing about this playground is that it refreshes itself every 10 minutes. You don't have to worry that you're breaking anything that people are actually counting on to work.

Now that we have a collection, we can save our collection for later. We can export it. This is just going to export it as a json file and now we can share it with our coworkers or whatever and they can very easily import it into their own collection. So we can choose the file. Here it is, and we'll just import it as a copy. Once you've done the work of creating these collections, it's very easy to share them with your team and that makes collaborating on the APIs very easy.

**Now that we've gone over the Postman tool and how to get information into the Postman tool and play with that information, we're ready to start really testing our APIs.**

**Chapter 3**

**Overview:**

**how to set up your Postman testing environment,**

**how we can use Postman to explore APIs**

**through the POISED testing strategy**

We'll be covering API testing strategy,

APIs

to be correct in that they are fulfilling their contract correctly.

provide good value for the developers creating applications.

intuitive to use.

We'll be using

the "POISED" heuristic to help us with our test coverage.

**P**arameters,

**O**utput,

**I**nterop,

**S**ecurity,

**E**rrors,

**D**ata.

**Before we start testing we need to make sure our test environment is set up properly.**

in Postman with our [Restful Booker Source](https://github.com/mwinteringham/restful-booker) collection.

There's two things to do before we really start exploring this API.

1 run a local copy of the Restful Booker app to test on.

You'll note that I'm using the publicly available API (https://restful-booker.herokuapp.com/ping) which is fine for a lot of testing, but we don't totally have control over what's happening. It resets every 10 minutes, and we can't see all the logs that are available.

**Restful Booker API locally, get code**

**https://github.com/mwinteringham/restful-booker**

**click [Clone or download] as a zip file.**

**click [Download]**

**TIP**

Restful Booker is a NODE application,

go to Nodejs.org

install the proper version for your OS - [NodeJS](https://nodejs.org/en/download) .

downloaded the code

unzip the package,

enter the directory,

run npm install

This will retrieve any other packages that Restful Booker needs.

the instructions need MongoDB, but actually do Not need to have MongoDB installed.

a couple of warnings for packages. We're not going to worry about that now. Everything's still going to run properly.

type npm start

go to <localhost:3001>, we can see that Restful Booker is running locally on our machines.

Currently, our collection is set up to access Restful Booker through the production url.

In order to add a new environment, the local environment, we go to the environment section of Postman.

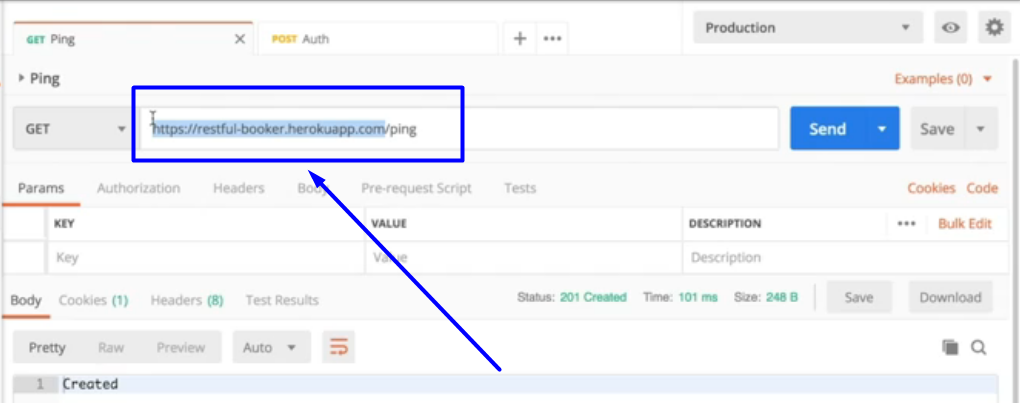
Currently we have nothing here, but if we manage environments;

create an environment (click the gear icon in the top right corner).

click [Add]

* **name** “Production.”
* **variable** rb\_url
  + - * to hold the name of the base url.
* **initial value** https://restful-booker .herokuapp.com

https://restful-booker .herokuapp.comverything is entered, simply click the “Add” button. Now that we've created our production environment, we want to set up our request to use the variable instead of this hardcoded url, because the point of the variable is that it can be inserted in various places.



What we're going to do is replace this part, with a double curly brace and then enter the name of our new variable:

{{rb\_url}}/ping

And you can test that you entered the right thing. If you hover over it, you'll see that it shows the Restful Booker Heroku app value that we want for production, and if we send this, you can see that it's working.

**Now that we have our production environment set up,**

**it's an easy matter to create a local environment as well.**

We go back to manage environments, and we are going to add a new one.

name: Local

variable rb\_url

initial value localhost:3001

click the “Add” button.

Now, when we switch this to the local environment, and we hover our variable, we'll see that it's pointing to local host. And when we send it, and we go to our local host, we can see that it did indeed do our local host instead of the production Heroku app.

Once that's done, update all of the requests with the new url and now we have a complete collection that will work in two different environments.

**The final thing we want to do is to make sure that our requests are authorized properly.**

If you'll remember the put (PUT), the patch (PATCH) and the delete (DEL), have an authorization requirement, in this case done by a cookie, but in reading the documentation, you can also do it through an authorization call.

Postman allows you to add authorization to all the requests in a collection.

next to collection, click **...**

click Edit

click Authorization.

Postman supports many authorization schemes.

use Basic Auth.

username “admin”

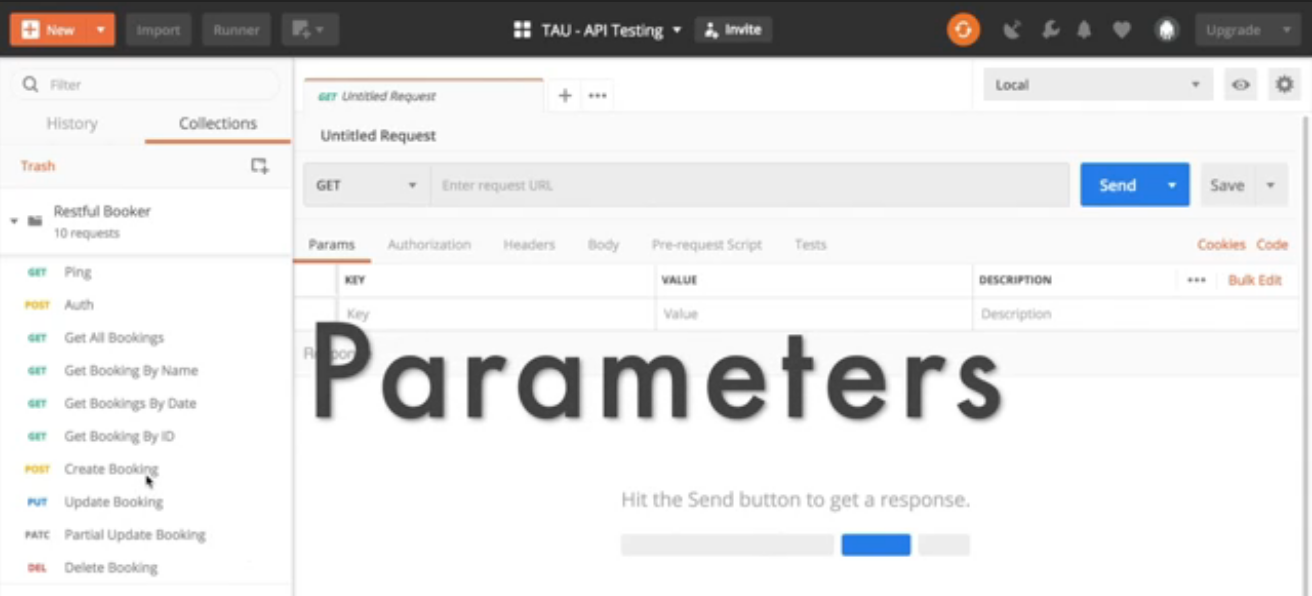
password “password123”

Now we update this.

Once we've set the authorization, we can just send our PUT request, and it's working. You'll see in the headers that there's an authorization header that was put in automatically by Postman.

So by having this authorization applied to all the requests in the collection, that means we can use the request without having to worry whether we have an up-to-date token or not.

**Now that we've created our production and local environments and edited an authorization header, we're ready to start going through our POISED heuristic to make sure that we're testing Restful Booker as completely as we can.**



***Parameters***

*O*

*I*

*S*

*E*

*D*

Let's look at one of these API's. Create Booking.

{

“firstname” : “Jim”,

“Lastname” : “Brown”,

“totalprice” : 111,

“depositpaid” : true,

“bookingdates” : {

“checkin” : “2018-01-01”,

“checkout” : “2019-01-01”

},

“additionalneeds” : “Breakfast”

}

Each of these is a parameter in the booking application. We want to make sure, first of all that the valid values work.

**classic boundary style testing on these parameters.**

i.e. **an empty string** in the first name parameter?

{

"firstname" : "",

"lastname" : "Brown",

"totalprice" : "111",

"depositpaid" : true,

"bookingdates" : {

"checkin" : "2018-01-01",

"checkout" : "2019-01-01"

},

"additionalneeds" : "Breakfast"

}

That causes an **error**.

What if it has spaces in it?

“firstname” : “dsfs sdfsd dfs”,

okay; does Not give an error.

Different parameters are going to act differently.

i.e. additional needs (additionalneeds) is empty:

“additionalneeds” : “”

That's fine.

What happens if the field is ***null***, what happens then?

“additionalneeds” : null

That's also okay.

What if we don't have this at all?

Deleting the parameter "additionalneeds" from the code still works without any errors.

Obviously the "additional needs" is completely **optional** and there's not really any controlling what's in it.

Whereas the "first name" parameter has to have some value.

What if it's just a single letter?

“firstname” : “J”,

Then that's fine as well.

learned that the "first name" has to be a non-empty string, whereas the "additional requests" value is completely optional.

**whether an invalid value causes a problem.**

There's **no type checking** in a JSON package (package.json) that's going through HTP.

Even though the "deposit paid" parameter is boolean,

I could easily make this a **string**.

It doesn't even have to be a string that matters.

It could just be **anything** and **we would get a valid result.**

“depositpaid” : “ksdjflskdjfsldkjsf”,

It could even be **a number**,

will work as well

we will get a result that says our "deposit was paid".

That's interesting.

What if it's an empty string?

“depositpaid” : “”,

causes an error.

We have all sorts of interesting things here going on with

**the deposit paid.**

A random string causes true,

an empty string causes an error.

**testing do on the parameters for your API's to find out how it reacts under different conditions.**